

# Statistical Relational Learning

Hendrik Blockeel<sup>1,2</sup>

<sup>1</sup>Katholieke Universiteit Leuven, Department of Computer Science

<sup>2</sup>Leiden Institute of Advanced Computer Science

**Abstract.** Relational learning refers to learning from data that have a complex structure. This structure may be either internal (a data instance may itself have a complex structure) or external (relationships between this instance and other data elements). Statistical relational learning refers to the use of statistical learning methods in a relational learning context, and the challenges involved in that. In this chapter we give an overview of statistical relational learning. We start with some motivating problems, and continue with a general description of the task of (statistical) relational learning and some of its more concrete forms (learning from graphs, learning from logical interpretations, learning from relational databases). Next, we discuss a number of approaches to relational learning, starting with symbolic (non-probabilistic) approaches, and moving on to numerical and probabilistic methods. Methods discussed include inductive logic programming, relational neural networks, and probabilistic logical or relational models.

## 1 Introduction

Machine learning approaches can be distinguished along a large number of dimensions. One can consider different *tasks*: classification, regression, and clustering are among the better known ones. One can also distinguish approaches according to what kind of *inputs* they can handle; the format of the *output* they produce; the algorithmic or mathematical description of the actual learning *method*; the *assumptions* made by that learning method (sometimes called its inductive bias); etc.

In this chapter, we will first have a closer look at the input format: what kind of input data can a learning system handle? This is closely related to properties of the learned model: this model is often a (predictive) function, which takes arguments of a particular type, and this type should be compatible with the input data. In this context, we will consider the *attribute-value learning* setting, which most machine learning systems use, and the *relational learning* setting, which is the setting in which statistical relational learning takes place. The distinction between attribute-value and relational learning is quite fundamental, and forms an important motivation for considering relational learning as a separate field. After discussing how relational learning is set apart from attribute-value learning (in Section 2), we will have a closer look at relational learning methods in general (Sections 3–4), and then zoom in on statistical relational learning (Section 5).

## 2 Relational learning versus attribute-value learning

### 2.1 Attribute-value learning

The term attribute-value learning (AVL) refers to a setting where the input data consists of a set of data elements, each of which is described by a fixed set of attributes, to which values are assigned. That is, the input data set  $D$  consists of elements  $\mathbf{x}_i$ ,  $i = 1, \dots, N$ , with  $N$  denoting the total number of elements in  $D$ . These elements are also called *examples*, *instances* or *individuals*. Each element is described by a number of *attributes*, which we usually denote  $A_i$ ,  $i = 1, \dots, n$ . Each attribute  $A$  has a set of possible values, called its *domain*, and denoted  $Dom(A)$ . The domains of the attributes may vary: an attribute  $A$  may be boolean, in which case its domain is  $Dom(A) = \{true, false\} = \mathbb{B}$ ;<sup>1</sup> it may be nominal, in which case its domain is a finite set of symbolic values  $\{v_1, v_2, \dots, v_{|Dom(A)|}\}$ ; it may be numerical, for instance  $Dom(A) = \mathbb{N}$  (discrete) or  $Dom(A) = \mathbb{R}$  (continuous); the domain may be some (totally or partially) ordered set; etc.

Thus, mathematically, the instances are points in an  $n$ -dimensional instance space  $\mathcal{X}$ , which is of the following form:

$$\mathcal{X} = Dom(A_1) \times \dots \times Dom(A_n)$$

We also call these points *n-tuples*.

Many learning systems, such as decision tree learners, rule learners, or instance-based learners, can handle this data format directly. Other learning approaches, including artificial neural networks and support vector machines, treat the instance space as a vector space, and assume  $Dom(A_i) = \mathbb{R}$  for all  $A_i$ . When the original data are not numerical, using the latter type of approaches requires encoding the data numerically. This can often be done in a relatively simple way (for instance, the boolean values true and false can be encoded as 1 and 0), and we will not discuss this in detail here.

For ease of discussion, let us now focus on the task of learning a classifier. Here, the instance space is typically of the form  $\mathcal{X} \times \mathcal{Y}$  with  $\mathcal{X} = Dom(A_1) \times \dots \times Dom(A_{n-1})$  and  $\mathcal{Y} = Dom(A_n)$ ; the instances are of the form  $(\mathbf{x}, y)$ , and the task is to learn a function  $f : \mathcal{X} \rightarrow \mathcal{Y}$  that, given some  $\mathbf{x}$ , predicts the corresponding  $y$ . If  $Dom(A_i) = \mathbb{R}$  for all  $A_i$ ,  $\mathbf{x}$  is an  $n - 1$ -dimensional vector; this means that we learn a function with  $n - 1$  input arguments, all of which are reals. This type of functions is well-understood, it is studied in great detail in calculus. Also when  $Dom(A_i) \neq \mathbb{R}$ , such functions are easy to define by referring to the different  $A_i$ . But the mathematical concept of a function is more general: we can also consider functions that take variable-length tuples, sets, sequences, or graphs as arguments. Such functions are generally much more difficult to specify. It is exactly that type of functions that we encounter in relational learning.

<sup>1</sup> We use  $\mathbb{B}$  for the set of booleans, analogously to the use of  $\mathbb{N}$  and  $\mathbb{R}$  for natural numbers and reals, respectively.

## 2.2 Relational learning

Relational learning refers to learning from data that have a complex structure, either internally or externally. A complex *internal structure*, in this case, implies that an instance cannot be described as a single point in a predefined  $n$ -dimensional space; rather, the instance consists of a variable number of components and relationships between them (e.g., a graph). *External structure* refers to the fact that relationships exist between the different data elements; the instances cannot be considered independent and identically distributed (i.i.d.), and properties of one instance may depend not just on other properties of itself, but also on properties of other instances somehow related to it.

We first give some motivating examples for both settings (internal or external structure); next, we discuss the connection between them, and how they compare to attribute-value learning. Much of the discussion in this subsection is based on Struyf and Blockeel [56].

**Learning from examples with external relationships** This setting considers learning from a set of examples where each example itself has a relatively simple description, for instance in the attribute-value format, and relationships may be present among these examples.

*Example 1.* Consider the task of web-page classification. Each web-page is described by a fixed set of attributes, such as a bag of words representation of the page. Web-pages may be related through hyper-links, and the class label of a given page typically depends on the labels of the pages to which it links.

*Example 2.* Consider the Internet Movie Database ([www.imdb.com](http://www.imdb.com)). Each movie is described by a fixed set of attributes, such as its title and genre. Movies are related to entities of other types, such as *Studio*, *Director*, *Producer*, and *Actor*, each of which is in turn described by a different set of attributes. Movies can also be related to each other via entities of other types. For example, they can be made by the same studio or star the same well known actor. The learning task in this domain could be, for instance, predicting the opening weekend box office receipts of the movies.

If relationships are present among examples, then the examples may not be independent and identically distributed (i.i.d.). Many learning algorithm assume they are, and when this assumption is violated, this can be detrimental to learning performance, as Jensen and Neville [31] show. On the other hand, knowledge about the relationships among examples can be beneficially exploited by the learning algorithm. Collective classification techniques [32], for example, take the class labels of related examples into account when classifying a new instance, which can lead to better predictive results.

Thus, to an attribute-value learner, relations between instances may hamper accurate learning, but to a relational learner, this relations are a source of information that can be put to good use.

**Learning from examples with a complex internal structure** In this setting, each example may have a complex internal structure, but no relationships exist that relate different examples to one another. Learning algorithms typically use individual-centered representations in this setting, such as logical interpretations [9] or strongly typed terms [42], which store all the data available about a given instance. Special cases of this setting include applications where the examples can be represented as graphs, trees, or sequences.

*Example 3.* Consider a database of candidate chemical compounds to be used in drugs. The molecular structure of each compound can be represented as a graph where the vertices are atoms and the edges are bonds. Each atom is labeled with its element type and the bonds can be single, double, triple, or aromatic bonds. Compounds are classified as active or inactive with regard to a given disease and the goal is to build models that are able to distinguish active from inactive compounds based on their molecular structure. Such models can, for instance, be used to gain insight in the common substructures, such as binding sites, that determine a compound’s activity.

**External or internal: It does not matter** In many applications, relationships are naturally viewed either as internal to instances, or external to them. If we need to classify nodes in a graph, for instance, since the instances to classify are individual nodes, the link structure is considered external. If we want to classify whole graphs, the link structure is internal. From a representation point of view, however, this difference does not matter. When we describe a single node to be classified, the node’s (external) context is part of its description. So regardless of whether the relational information is internal or external to the object being described, it is always internal to the representation of the object.

Thus, from the point of view of representation of the input data, there is really only one relational learning setting (even if many different tasks can be defined in this setting). This setting is not equivalent to the attribute-value learning setting, however. There is no general way in which all the information available in a relational dataset can be mapped to a finite set of attributes (thus making attribute-value learning possible) without losing information. Relational learning cannot be reduced to attribute-value learning; it is inherently more difficult than the latter. In the next section we argue in more detail why this is the case.

### 2.3 Mapping relational data to attribute-value data

As a concrete example of relational learning, consider the case where the function to be learned takes graphs as input, and produces a boolean output. An example of this setting is the pharmacophore problem discussed later on. Assume that the target function is of the form: “if the graph contains a subgraph isomorphic to  $G$ , then it is positive, otherwise negative”, with  $G$  a particular graph. For brevity, we refer to this target function as  $\text{CONTAINS-}G$ , and we call the class of

all such functions  $\text{CONTAINS-}\mathcal{G}$ . We use  $\mathcal{G}$  to denote the “set of all finite graphs” (more formally, the set of all graphs whose node set is a finite subset of  $\mathbb{N}$ ; each imaginable finite graph is then isomorphic to a graph in  $\mathcal{G}$ ).

In the attribute-value framework, we assume that objects  $\mathbf{x}$  are described by listing the values of a fixed set of attributes. If the object descriptions are not given in this format, the question arises whether it is possible to represent them with a fixed set of attributes such that any function definable on the original representation can be defined on the attribute-value representation. That is, we need to find a mapping  $p$  from the original description space  $\mathcal{X}$  to a product space  $\mathcal{X}'$  such that, given a class of functions  $\mathcal{F}$  from  $\mathcal{X}$  to  $\mathbb{B}$ , for all  $f \in \mathcal{F}$  there is a corresponding  $f' : \mathcal{X}' \rightarrow \mathbb{B}$  such that  $f(\mathbf{x}) = f'(p(\mathbf{x}))$ . In other words, applying  $f'$  to the attribute value representation of the objects gives the same result as applying  $f$  to the original representation.

Clearly, such a mapping should be injective; that is, it should not be the case that two different objects ( $\mathbf{x}_1 \neq \mathbf{x}_2$ ) are mapped to the same representation  $\mathbf{x}'$ . If that were the case, then any function  $f$  for which  $f(\mathbf{x}_1) \neq f(\mathbf{x}_2)$  cannot possibly have an equivalent function  $f'$  in the attribute-value space.

Concretely, for the class of functions  $\text{CONTAINS-}\mathcal{G}$ , we would need an attribute-value representation where for each  $G \in \mathcal{G}$ , the condition “contains  $G$  as a subgraph” can be expressed in terms of the attributes. This is trivially possible if we make sure that for each  $G$ , a boolean attribute is defined that is true for a graph that contains  $G$ , and false otherwise. The problem is that there is an infinite number of such graphs  $G$ , hence, we would need an infinite number of such attributes. If the size of  $G$  is limited to some maximum number of nodes or edges, the number of different options for  $G$  is finite, but it can still be a huge number, to the extent that it may be practically impossible to represent graphs in this way.

Besides the trivial choice of attributes mentioned above, one can think of other attributes describing graphs, such as the number of nodes, the number of edges, the maximal degree of any node, and so on. The question is, is there a finite set of attributes such that two different graphs are never mapped onto the same tuple? When all these attributes have finite domains, this is clearly not the case: the number of different tuples is then finite, while the number of graphs is infinite, so the mapping can never be injective. If some attributes can have infinite domains, however, an injective mapping can be conceived. The set of all finite graphs is enumerable, which means a one-to-one mapping from graphs to the natural numbers exists. Hence, a single attribute with domain  $\mathbb{N}$  suffices to encode any set of graphs in a single table without loss of information. Thus, strictly speaking, a one-to-one encoding from the set of graphs to attribute-value format exists. But now, the problem is that a condition of the form “has  $G$  as a subgraph”, for a particular  $G$ , can not necessarily be expressed in terms of this one attribute in a way that might be learnable by an attribute-value learner. Attribute-value learners typically learn models that can be expressed as a function of the inputs using a limited number of mathematical operations. Neural networks, for instance, learn functions that can be expressed in terms of

the input attributes using summation, multiplication, and application of a non-linear squashing function. Suppose that, for instance, for a particular encoding, the set of graphs containing graph 32 as a subgraph is the infinite set  $S_{32} = \{282, 5929, 11292, \dots\}$ . It is not obvious that a formula exists that uses only the number 32, the operators  $+$  and  $\times$ , a sigmoid function and a threshold function, and that results in true for exactly the numbers in the set (and false otherwise), let alone that such an expression could be found that works for each graph number  $n$  and the corresponding set  $S_n$ .

Informally, we call a learning problem *reducible to attribute-value learning* if an encoding is known (a transformation from the original space  $\mathcal{X}$  to a product space  $\mathcal{X}'$ ) such that an existing attribute-value learner exists that can express for each function  $f : \mathcal{X} \rightarrow \mathbb{B}$  the corresponding  $f' : \mathcal{X}' \rightarrow \mathbb{B}$ .

Reducibility to AVL implies that a (relational) learning problem can be transformed into an attribute value learning problem, after which a standard attribute-value learner can be used to solve it. Many problems, including that of learning a target function in  $\text{CONTAINS-}\mathcal{G}$ , are not reducible to AVL. Generally, problems involving learning from data where data elements contain sets (this includes graphs, as these are defined using sets of nodes and edges) are not reducible to AVL.

**Propositionalization versus relational learning** Among relational learners, we can distinguish systems that use so-called *propositionalization* as a preprocessing step, from learners that do not. The latter could be considered “truly” relational learners.

Propositionalization refers to the mapping of relational information onto an attribute-value (also known as “propositional”) representation. This is often done in a separate phase that precedes the actual learning process. Propositionalization amounts to explicitly defining a set of features, and representing the data using that set. Formally, given a data set  $D \subseteq \mathcal{X}$ , the process of propositionalization consists of defining a set  $F$  of features  $\phi_i : \mathcal{X} \rightarrow R_i$ , where each  $R_i$  is nominal or numerical, and representing  $D$  using these features as attributes, i.e., representing  $D$  by  $D' = \{(\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \dots, \phi_n(\mathbf{x})) | \mathbf{x} \in D\}$ . We call  $D'$  a propositional representation of  $D$ .

The feature set may be fixed for a particular algorithm, it may be variable and specified by the user, or it may be the result of some feature selection process that, from an initial set of features, retains only those that are likely to be relevant for the learning task.

When a learning problem is not reducible to AVL, then, no matter how the features are defined, propositionalization causes loss of information: some classifiers expressible in the original space may no longer be expressible in the feature space.

Truly relational learners do not suffer from this problem: they learn a function in the original space. For instance, consider again our example of learning functions in  $\text{CONTAINS-}\mathcal{G}$ . A truly relational learner uses the original graph representations and can express any function in  $\text{CONTAINS-}\mathcal{G}$ .

As argued earlier, in principle, we could define one feature for each  $G \in \mathcal{G}$ , which expresses whether the graph being described contains  $G$  as a subgraph (but we would need an infinite number of features for this). More generally, whatever the class of functions is that a relational learner uses, we could define for each function in that class a feature that shows the result of that function, when applied to a particular instance. Thus, each relational learner could be said to implicitly use a (possibly infinite) set of features. From this point of view, the main difference between a truly relational learner and a propositional learner (or a learner that uses propositionalization) is that a truly relational learner typically proceeds by gradually selecting more and more actual features from some initial set of potential features. The result of the relational learning is a function that can be described in terms of a small number of relevant features, and which is constant in all other. While the set of relevant features is finite, the number of features that can in principle be considered for inclusion may be infinite.

Note that propositionalization-based systems may also perform feature selection, just like truly relational learners. The difference is that a propositionalization-based system first constructs a representation of the data set in terms of a finite set of features, then reduces that set. A truly relational learner does not construct the propositionalized data set  $D'$  at any point.

Relational learners are more powerful in the sense that they can use a much larger feature set: even if not infinite, the cardinality of this set can easily be so high that even storing the values of all these features is not feasible, making propositionalization intractable. On the other hand, they necessarily search only a very small (and always finite) part of this space, which means that good heuristics are needed to steer the search, in order to ensure that all relevant features are indeed encountered.

Propositionalization is a step that is very often used in practice, when using machine learning on relational data. In fact, practitioners often assume implicitly that such a propositionalization step is necessary before one can learn. That is correct if one considers attribute-value learners only, but false when relational learners are also an option.

## 2.4 Summary of this section

Most off-the-shelf learning systems assume the input data to be in the attribute-value format (sometimes also called the “standard” format). Relational data cannot be represented in the attribute-value format without loss of information. There are then two options: either the user converts the data into attribute-value format and accepts information loss, or a learning algorithm must be used that handles such relational data directly. Such a relational learner could be said to ultimately construct an attribute-value representation just as well, either explicitly (through propositionalization) or implicitly (in which case it may be searching a huge space of potential features for the most informative ones). The last type of learner is the most powerful, but also faces the most challenging task.

### 3 Relational learning: tasks and formalisms

Many different kinds of learning tasks have been defined in relational learning, and an even larger number of approaches have been proposed for tackling these tasks. We give an overview of different learning settings and tasks that can be considered instances of relational learning. Where mentioning methods, we focus on symbolic and non-probabilistic methods; methods based on neural processing and probabilistic inference are treated in more detail in the next two sections.

#### 3.1 Inductive Logic Programming

In inductive logic programming (ILP), the input and output knowledge of a learner are described in (variants of) first-order predicate logic. Languages based on first-order logic are highly expressive from the point of view of knowledge representation, and indeed, a language such as Prolog [2] can be used directly to represent objects and the relationships between them, as well as background knowledge that one may have about the domain.

*Example 4.* This example is based on the work by Finn et al. [21]. Consider a data set that describes chemical compounds. The active compounds in the set are ACE inhibitors, which are used in treatments for hypertension. The molecular structure of the compounds is represented as a set of Prolog facts, such as:

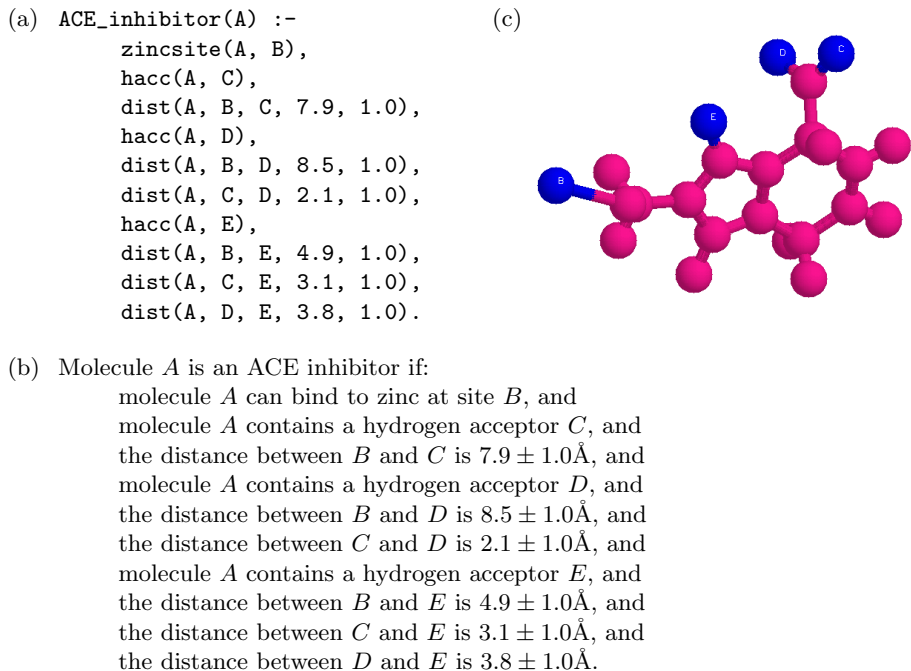
```
atom(m1, a1, o).  
atom(m1, a2, c).  
...  
bond(m1, a1, a2, 1).  
...  
coord(m1, a1, 5.91, -2.44, 1.79).  
coord(m1, a2, 0.57, -2.77, 0.33).  
...
```

which states that molecule `m1` includes an oxygen atom `a1` and a carbon atom `a2` that are single bonded. The `coord/5` predicate lists the 3D coordinates of the atoms in the given conformer. Background knowledge, such as the concepts zinc site, hydrogen donor, and the distance between atoms, are defined by means of Prolog clauses. Fig. 1 shows a clause learned by the inductive logic programming system PROGOL ([18], Ch. 7) that makes use of these background knowledge predicates. This clause is the description of a pharmacophore, that is, a submolecular structure that causes a certain observable property of a molecule.

Note that, in Prolog, variables start with capitals, and constants with lowercase characters. We will use this convention also when writing logical clauses outside the Prolog context.

Research in inductive logic programming originally focused on concept learning. Concept learning, which is often considered a central task in artificial intelligence and was for a long time the main focus of machine learning research,





**Fig. 1.** (a) Prolog clause modeling the concept of an ACE inhibitor in terms of the background knowledge predicates `zincsite/2`, `hacc/2`, and `dist/5`. The inductive logic programming system PROGOL automatically translates (a) into the “Sternberg English” rule (b), which can be easily read by human experts. (c) A molecule with the active site indicated by the dark colored atoms. (Example based on Finn et al. [21].)

concerns learning a definition of a concept from example instances. In the ILP setting, the concept to be learned is an  $n$ -ary relation or predicate, defined intentionally by a set of rules, and the task is to discover this set of rules by analyzing positive and examples, which are tuples said (not) to belong to the relation. The example in Fig. 1 is a concept learning task: an operational definition of the concept of an ACE-inhibitor is learned.

Pioneering work on concept learning in the first order logic context resulted in well-known ILP systems such as FOIL[51] and PROGOL [45]. Later, the focus has widened to include many other tasks such as clausal discovery, where the goal is to discover logical clauses that hold in a dataset, without focusing on clauses that define a particular concept[11]; regression, where clauses are learned that compute a numerical prediction [33]; frequent pattern discovery [16]; and reinforcement learning [17,57]. Also additional learning paradigms, besides rule learning, have been explored, including decision trees, instance-based learners, etc. Extensive overviews of the theory of inductive logic programming, including descriptions of tasks, methods, and algorithms, are available in the literature [10,18].

### 3.2 Learning from Graphs

A graph is a mathematical structure consisting of a set of nodes  $V$  and a set of edges  $E \subseteq V \times V$  between those nodes. The set of edges is by definition a binary relation defined over the nodes. Hence, for any learning problem where the relationships between examples can be described using a single binary relation, the training set can be represented straightforwardly as a graph. This setting covers a wide range of relational learning tasks, for example, web mining (the set of links between pages is a binary relation), social network analysis (binary “friend” relation), etc. Non-binary relationships can be represented as hypergraphs; in a hypergraph, edges are defined as subsets of  $V$  of arbitrary size, rather than elements of  $V \times V$ .

In graph-based learning systems, there is a clear distinction between approaches that learn from examples with external relationships, where the whole data set is represented as a single graph and each node is an example, and individual-centered approaches, where each example by itself is a graph. In the first kind of approaches, the goal is often to predict properties of existing nodes or edges, to predict the existence or non-existence of edges (“link discovery”), to predict whether two nodes actually refer to the same object (“node identification”), detection of subgraphs that frequently occur in the graph, etc. When learning from multiple graphs, a typical goal is to learn a model for classifying the graphs, to find frequent substructures (where frequency is defined as the number of graphs a subgraphs occurs in), etc.

Compared to other methods for relational learning, graph-based methods typically focus more on the structure of the graph, and less on properties of single nodes. They may take node and edge labels into account, but often do not allow for more elaborate information to be associated with each node.

Graph mining methods are often more efficient than other relational mining methods because they avoid certain kinds of overhead, but are typically still NP-complete, as they generally rely on subgraph isomorphism testing. Nevertheless, researchers have been able to significantly improve efficiency or even avoid NP-completeness by looking only for linear or tree-shaped patterns, or by restricting the graphs analyzed to a relatively broad subclass. As an example, Horvath et al. [29] show that a large majority of molecules belong to the class of outerplanar graphs, and propose an efficient algorithm for subgraph isomorphism testing in this class.

Well-known systems for graph mining include gSpan [71], Gaston [47], and Subdue [6]. Excellent overviews of the field are provided by Cook and Holder [7] and Washio and Motoda [70].

### 3.3 Multi-Relational Data Mining

Multi-relational data mining approaches relational learning from the relational database point of view. The term “multi-relational” refers to the fact that from the database perspective, one learns from information spread over multiple tables

or relations, as opposed to attribute-value learning, where one learns from a single table.

Multi-relational data mining systems tightly integrate with relational databases. Mainly rule and decision tree learners have been developed in this setting. Because practical relational databases may be huge, most of these systems pay much attention to efficiency and scalability, and use techniques such as sampling and pre-computation (e.g., materializing views). An example of a scalable and efficient multi-relational rule learning system is CrossMine [72].

In the context of multi-relational data mining, propositionalization boils down to summarizing all the data relevant to a single instance, which may be spread over multiple tuples in multiple relations, into a single tuple. As explained before, this is in general not possible without loss of generalization. Krogel et al. [39] compare a number of methods for propositionalization in the context of multi-relational learning.

Most inductive logic programming systems are directly applicable to multi-relational data mining by representing each relational table as a predicate. This is possible because the relational representation is essentially a subset of first-order logic (known as datalog). Much research on multi-relational data mining originates within the ILP community [18]. Nevertheless, there is a clear difference between ILP and multi-relational learning in terms of typical biases of these methods, as will become clear in the next section.

## 4 Neural network based approaches to relational learning

Among the many approaches to relational learning, a few neural network approaches have been proposed. We briefly summarize two of them here, and discuss a third one in more detail.

### 4.1 CIL<sup>2</sup>P

The KBANN system (Knowledge-Based Artificial Neural Networks) [60] was one of the first to integrate logical and neural representations and inference. It used propositional logic only, however, and hence cannot be considered a truly relational learning system. Perhaps the earliest system that did combine first-order logic with neural networks, is CIL<sup>2</sup>P, which stands for Connectionist Inductive Learning and Logic Programming [8]. This system is set in the first order logic context, and aims at integrating neural network inference and logic programming inference, both deductive (using the model to draw conclusions) and inductive (learning the model). This integration makes it possible to learn logic programs using neural network learning methods.

A limitation of this approach is that the neural network represents a ground version of the logic program.<sup>2</sup> As the grounding of a first order logic program

---

<sup>2</sup> Given a logic program, grounding it means replacing every rule that contains variables with all the possible instantiations (i.e., applications to constants) of that rule; the resulting program is equivalent to the original one.

may be large, the neural network is correspondingly large and may be inefficient because of this. Note, in particular, that a major advantage of a logic programming language such as Prolog is that it can reason on the level of variables, without instantiating those variables (a type of inference that in statistical relational learning is called *lifted inference*). For instance, if a rule  $p(X) \leftarrow q(X, Y)$  is known, as well as a rule  $q(X, a)$ , Prolog can deduce that  $p(X)$  is true whatever  $p$  is, without needing to prove this separately for each concrete case. A neural network that essentially operates on the level of ground instances must make this inference separately for each case, which makes it complex and inefficient.

The approach that will be discussed next, avoids the construction of a large ground network (i.e., a network the size of which depends on the size of the logic program’s grounding), and at the same time aims at constructing a broader type of models than is typically considered when using logic-based representations.

## 4.2 Relational neural networks

Relational Neural Networks (RNNs) [62] are a neural network based approach to relational learning that is set in the context of multi-relational data mining. They were originally proposed to reconcile two rather different biases of relational learners; these biases have been called selection bias and aggregation bias [1], and they are related to how sets are handled by the system. We will first describe the motivation for the development of RNNs in more detail.

### Motivation: aggregation-oriented versus selection-oriented methods

The main problem with relational data is the existence of one-to-many relationships. We may need to classify an object  $\mathbf{x}$ , taking into account properties of other objects that it is related to via a particular relation, and there may be multiple such objects. If  $S(\mathbf{x})$  is the set of these objects, then the question is what features we define on this set. As argued before, defining features for sets is a non-trivial task (learning from sets is not reducible to AVL).

As it turns out, within relational learning, we can consider two quite different approaches. Both of these learn features of the type  $\mathcal{F}(\sigma_C(S(\mathbf{x})))$ , where  $\sigma_C$  is the selection operator from relational algebra, and  $\mathcal{F}$  is an aggregation function, which summarizes a set into a single value (for instance,  $\mathcal{F}$  could be the average, maximum, minimum or variance of a set of reals; it could be the mode of a set of nominal values; it could be the cardinality of any set; etc.; it could also be any combination of these).

In the **first type of learner**,  $S(\mathbf{x})$  is defined in a straightforward way; it could be, for instance, all objects  $\mathbf{y}$  related to  $\mathbf{x}$  through the relation  $R(\mathbf{x}, \mathbf{y})$ . This  $S$  is then summarized using a small set of features, each of which is one aggregation function (often just the standard functions are used, such as count, max, min, average). Sometimes the set is viewed as a sample from a distribution, and this distribution’s parameters or other characteristic numbers are used to describe  $S$ . (For instance, besides the mean and variance, one could also estimate higher order moments of the distribution; the  $k$ -th moment of a distribution  $p$

is defined as  $E(x^k) = \int_x x^k p(x) dx$ .) This type of features has been considered by several researchers in relational learning; for instance, standard aggregation functions are used to represent sets of related objects in PRMs (probabilistic relational models, see next section) or used as features in the propositionalization method used by Krogel and Wrobel [40], whereas Perlich and Provost [48] consider aggregation functions based on a broader set of distributional characteristics.

The **second type of learner** we consider here is typical for inductive logic programming. When a clause is learned, such as  $p(X) \leftarrow r(X, Y), s(Y, a)$ , the set of objects  $\mathbf{y}$  to which  $\mathbf{x}$  is related is defined in a relatively complex way; it is a selection of tuples from the cartesian product of all relations mentioned in the body of the clause. The set of instantiations of  $Y$  that are obtained for a particular value of  $X$  can be written in relational algebra as  $\sigma_{S.A2=a' \wedge R.A2=S.A1}(R \times S)$ .<sup>3</sup> The clause body evaluates to true or false depending on whether this set of instantiations is empty or not. Thus, the features expressed by such a clause can be written as  $\mathcal{F}_{\exists}(\sigma_C(R_1 \times \dots \times R_k))$ , where  $\mathcal{F}_{\exists}$  represents the existential aggregate function (which returns true if its argument is non-empty, and false otherwise), the  $R_i$  are all the relations mentioned in the clause body, and  $C$  expresses the conditions explicitly or implicitly imposed on the variables in the clause.

We now see that ILP systems typically construct relatively complex selection conditions, but in the end simply test for emptiness of the resulting set. We call them **selection-oriented systems**: they invest effort in constructing a good selection condition, but ignore the fact that multiple aggregation functions might be useful for characterizing the resulting set. The first type of approaches we just mentioned, which we call **aggregation-oriented systems**, do the opposite: they consider the possible usefulness of multiple aggregation functions, but do not invest effort in building a more complex set  $S(\mathbf{x})$  than what can be defined using a single relation, without further selection of the elements in the relation.

To illustrate this situation in somewhat more concrete terms: suppose a person is to be classified based on information about their children; one approach could consider the number of children, or their average age; another approach could consider whether the person has any daughters (assuming daughters to be those element in the children relation for which the attribute Sex has the value ‘female’); but none of the mentioned approaches can consider, as a feature, the age of the oldest daughter.

Progress towards resolving this issue was presented by Vens et al. [69], who show how the systematic search performed by an ILP system can be extended towards clauses that can contain aggregation functions on conjunctions of literals. This essentially solves the problem of learning features of the form  $\mathcal{F}(\sigma_C(S(\mathbf{x})))$ , where both  $C$  and  $\mathcal{F}$  are non-trivial, for multiple standard aggregation functions (max, min, average, count). But the problem remains that  $\mathcal{F}$  is restricted to a fairly small set of aggregation functions. Combinations of the results of these functions can be constructed afterwards by means of standard machine learning

<sup>3</sup> As arguments of logical predicates have no fixed name, but attributes in relational algebra do, we use  $A_i$  to refer to the  $i$ ’th argument of any predicate.

techniques, but any aggregation function that cannot be described as a combination of these basic building blocks, remains unlearnable.

**Relational neural networks** The main motivation for learning relational neural networks was the desire to have a learner that learns features of the form  $\mathcal{F}(\sigma_C(S(\mathbf{x})))$  with possibly complex  $\mathcal{F}$  and  $C$ , though not necessarily in an explicit form. In the same way that a neural network can approximate many functions without building the actual expression that defines these functions, such relational neural networks should be able to approximate any feature of the form  $\mathcal{F}(\sigma_C(S(\mathbf{x})))$ , without being restricted to combinations of predefined aggregation functions, and without any restrictions regarding the form of  $C$ .

Since  $S(\mathbf{x})$  is an unbounded set of tuples, such a neural network should be able to handle an unbounded set as a single input unit, and provide one or more numerical outputs for that set. To achieve this aim, Uwents and Blockeel [62] use recurrent neural networks, to which the elements of the set are presented one by one. As a recurrent neural network processes a sequence of elements, rather than an unordered set, its output may depend on the order in which the elements of the set are presented. To counter this effect, reshuffling is used; that is, each time a set is presented to the network during training, its elements are presented to it in a random sequence. This implies that the network is forced to become as order-independent as possible; however, it remains in essence a sequence processing system, and complete order-independence is not necessarily achievable. Uwents and Blockeel experimented with multiple architectures for recurrent neural networks, including standard architectures such as the Jordan architecture [62], but also cascade-correlation networks [64], and found that some of these could learn relatively complex features quite well.

In a more extensive comparison [63,61], a toy dataset about classification of trains into eastbound and westbound is used, in which artificial target concepts of varying complexity are incorporated. One of the more complicated concepts is: “Trains having more than 45 wheels in total and at least 10 rectangle loads and maximum 27 cars are eastbound, the others are westbound”. Note that trains are represented as sequences of cars, where each car has a number of properties listed; properties such as the total number of cars or wheels in the train, or the number of “cars with rectangle loads” are not represented explicitly, but need to be constructed through aggregation and selection. The comparison showed the cascade-correlation approach to work better than other neural network based approaches; it achieves near-perfect performance on simple concepts, and better performance on more complex concepts than state-of-the-art learners that are propositionalization-based [40] or try to learn a symbolic representation of the concept [66].

### 4.3 Graph neural networks

Graph neural networks (GNNs) are discussed elsewhere in this volume, and we refer to that chapter for more details on the formalism. We limit ourselves here to

pointing out some of the main differences between relational neural networks and graph neural networks. First, the setting is different: consistent with the naming, RNNs are set in the context of relational databases, whereas GNNs are set in the context of graphs. While there is a connection between these two formalisms, there are also obvious differences. Consider a tuple in a relational database as a node in a graph, with foreign key relationships defining the edges in the graph. Since tuples can come from different relations, the nodes in this graph are typed. RNNs naturally define a separate model per type of node, whereas GNNs define the same model for all nodes. (RNN behavior can of course be simulated by introducing an attribute for each node that represents its type; the value of that attribute would then be used in the GNN model.) Further, RNNs aim more explicitly at approximating a particular type of features, and have been evaluated mostly in that context. GNNs work with undirected graphs, and as such create a more symmetric model. Uwents et al. [65] provide an extensive discussion of the relationship between GNNs and RNNs, as well as an experimental comparison.

## 5 Statistical relational learning

The above approaches to relational learning do not rely strongly on probability theory. Although many types of models, and their predictions, can be interpreted in a probabilistic manner (for instance, when a rule has a coverage of 15 positives and 5 negatives, it might be said to predict positive with 75% certainty), they do not necessarily define a unique probability distribution, and inference in these models is not based on probability theory. In statistical relational learning, the focus will be on models that by definition define a probability distribution, such as probabilistic graphical models (which includes Bayesian networks and Markov networks).

In the following we discuss a number of approaches to statistical relational learning. We start with approaches set in the context of graphical models; next, we discuss approaches set in the context of relational databases, and finally, of first order logic.

There is a plethora of alternative approaches to statistical relational learning, among which the relationships are not always clear. Many approaches seem to be differing mostly in syntax, yet there are often subtle differences in how easily certain knowledge can be expressed. We do not aim at giving an exhaustive overview here, or at indicating in exactly what way these methods differ from each other. Rather, we will discuss a few representative methods in detail.

### 5.1 Structuring graphical models

**Graphical models** Probabilistic graphical models define a joint distribution over multiple random variables in a compact way. They consist of a graph that implies certain independence relations over the variables. These independence relations imply that the joint distribution can be written as a product of a number of lower-dimensional factors. The graphical model consists of a graph

together with these factors; the graph imposes a certain structure of the joint distribution, while the factors determine it uniquely.

The best-known examples of graphical models are Bayesian networks and Markov networks. In both cases, there is one node for each random variable. Bayesian networks use directed acyclic graphs (DAGs); with each node one factor is associated, and that factor is equal to the conditional probability distribution of the node given its parents. Markov networks use undirected graphs; here, one factor is associated with each maximal clique in the graph.

When learning graphical models, a distinction can be made between *structure learning*, which involves learning the graph structure of a graphical model, and *parameter learning*, which involves learning the factors associated with the graph structure.

In both cases, a distinction can be made between generative and discriminative learning. This difference is relevant when it is known in advance what variables will need to be predicted (the so-called target variables). Let  $\mathbf{Y}$  denote the set of target variables, and  $\mathbf{X}$  the set of all other variables that are not target variables. The task is to learn a predictive model that predicts  $\mathbf{Y}$  from  $\mathbf{X}$ . In the probabilistic setting, one predicts not necessarily specific values for the variables in  $\mathbf{Y}$ , but a probability distribution for  $\mathbf{Y}$ . Given an observation  $\mathbf{x}$  of  $\mathbf{X}$ , the probability that  $\mathbf{Y} = \mathbf{y}$  (or, more generally, the probability density for  $\mathbf{y}$ ) is then  $p_{\mathbf{Y}|\mathbf{X}}(\mathbf{x}, \mathbf{y})$ . The direct approach to predictive learning consists of learning a model of  $p_{\mathbf{Y}|\mathbf{X}}$ ; this is called discriminative learning. An indirect approach consists of learning the joint distribution  $p_{\mathbf{X}, \mathbf{Y}}$ . All other distributions can be derived from this joint distribution, including the conditional distribution of  $\mathbf{Y}$  given  $\mathbf{X}$ :  $p_{\mathbf{Y}|\mathbf{X}}(\mathbf{x}, \mathbf{y}) = p_{\mathbf{X}, \mathbf{Y}}(\mathbf{x}, \mathbf{y}) / p_{\mathbf{X}}(\mathbf{x})$  with  $p_{\mathbf{X}}(\mathbf{x}) = \int_{\mathbf{y}} p_{\mathbf{X}, \mathbf{Y}}(\mathbf{x}, \mathbf{y}) d\mathbf{y}$ . This indirect approach is called generative learning.

Generative learning is the most general approach; it does not require the target variables  $\mathbf{Y}$  to be specified in advance, in contrast to discriminative learning. On the other hand, discriminative learning can be more efficient and more accurate because it focuses directly on the task at hand.

We refer to [37] for a more detailed introduction to graphical models.

**Dynamic Bayesian Networks** Until now, we have simply assumed that a set of variables is given, without any structure on this set, except possibly for the fact that some variables are considered observed (their values will be given, when using the model) and other unobserved (their values will need to be predicted using the model), and this partitioning may be known in advance.

In many cases, there is more structure in the variables. A typical case is when a dynamic process is described: the values of the variables change over time, and we can talk about the value of a variable  $V_i$  at timepoint  $0, 1, 2, \dots$ . Let us denote with  $V_i^{(t)}$  the variable that represents the value of variable  $V_i$  at time point  $t$ ; we assume discrete time points  $t \geq 0$ . While the state of the system changes over time, its dynamics are constant: for instance, the value of  $V_i^{(t)}$  depends on  $V_i^{(t-1)}$  (or more generally on  $V_j^{(t-k)}$  for any  $k$ ) in the same way, for any  $t > 0$ .



As an example of this, consider a hidden Markov model (HMM). Such a model describes a system that at any time is in one particular state, from a given set of states  $S$ . Its current state depends probabilistically on its previous state; that is, for any pair of states  $(s, s')$ , there is a fixed probability that a system that is in state  $s$  at time  $t$ , will be in state  $s'$  at time  $t + 1$ . Further, while the state itself is unobservable, at each time point the value of a particular variable  $X$  can be observed, and the value of  $X$  depends probabilistically on the current state.

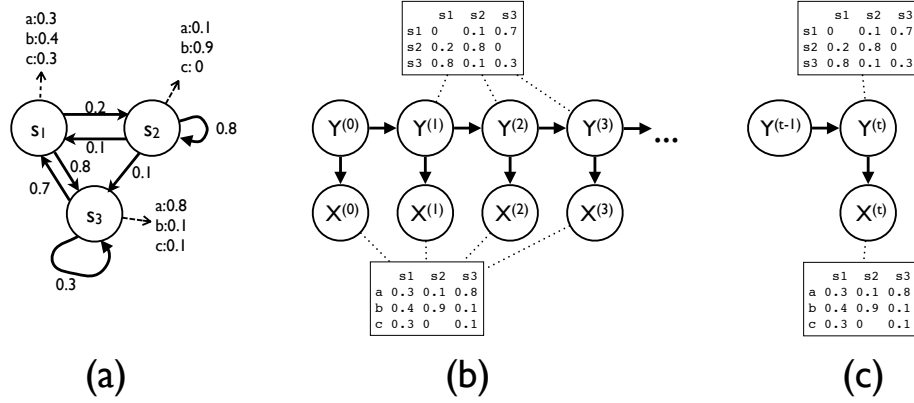
Keeping our earlier convention of using  $\mathbf{X}$  for the set of observed variables, and  $\mathbf{Y}$  for the set of unobserved variables, we use  $X^{(t)}$  to denote the output value at time  $t$  and  $Y^{(t)}$  to denote the state at time  $t$ ; we then have  $\mathbf{X} = \{X^{(0)}, X^{(1)}, X^{(2)}, \dots\}$  and  $\mathbf{Y} = \{Y^{(0)}, Y^{(1)}, Y^{(2)}, \dots\}$ . We can express the assumptions of a HMM in a graphical model by stating that, for all  $t > 0$ ,  $Y^{(t)}$  depends on  $Y^{(t-1)}$  (and this dependency is the same for all  $t$ ), and for all  $t \geq 0$ ,  $X^{(t)}$  depends on  $Y^{(t)}$  (again, in the same way for all  $t$ ).

Figure 2 shows an example of a hidden Markov model (a), and how it is modeled as an (infinite) Bayesian network (b). Because the dependencies of  $Y^{(t)}$  on  $Y^{(t-1)}$  and of  $X^{(t)}$  on  $Y^{(t)}$  are the same, we can express the Bayesian network more compactly by just showing for one particular time slice  $t$  how  $Y^{(t)}$  and  $X^{(t)}$  depend on other variables.

HMMs are a special case of dynamic Bayesian networks (DBNs). In a DBN, we also have variables  $V_i^{(t)}$ , but there is more flexibility with respect to which variables are observed and which are not, and with respect to the dependencies between variables. Generally, a variable  $V_i^{(t)}$  can depend on any variable  $V_j^{(t-k)}$  for any  $k \geq 0$ , as long as the set of dependencies form a sound Bayesian network (i.e., there are no cyclic dependencies). The DBN can be represented by a standard Bayesian network that contains as many variables, spread over as large a time window, as needed.

Dynamic Bayesian networks are an example of directed models with a repeating structure. Similarly, undirected models with structure can be defined. A well-known example of such models are conditional random fields (CRFs) [41]. For details on these, we refer to the literature.

**Plates** Within the graphical model community, *plates* have been introduced as a way of structuring Bayesian networks [5,55]. A plate indicates a substructure that is actually repeated multiple times within the graph. More precisely, several isomorphic copies of the substructure occur in the graph, and the factors associated with these copies are the same, modulo renaming of nodes. An example is shown in Figure 3. There is an arc from  $X$  to  $Y$ , but by drawing a rectangle around  $Y$  we indicate that there are actually multiple variables  $Y_i$ . The conditional probability distribution of  $Y_i$  given  $X$  is the same for all  $Y_i$ . Note that, because of this, plates are not only a way of writing the graph more compactly; they add expressiveness, as they allow us to impose an additional constraint on the factorization. Without plate notation, one could indicate that  $Pr(X, Y_1, Y_2, Y_3) = Pr(X)Pr(Y_1|X)Pr(Y_2|X)Pr(Y_3|X)$ , but not that, in ad-



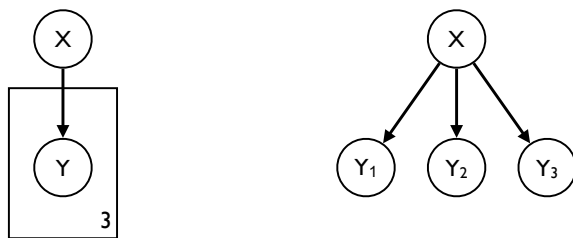
**Fig. 2.** (a) A schematic representation of a Hidden Markov model. (b) An infinite Bayesian network representing the same model, with the conditional probability tables of  $X^{(t)}$  and  $Y^{(t)}$  shown. Since each  $X^{(t)}$  and  $Y^{(t)}$  has the same dependencies, the tables are shown only once. (c) The same network, represented more compactly as a dynamic Bayesian network.

dition,  $Pr(Y_1|X) = Pr(Y_2|X) = Pr(Y_3|X)$ . With a plate model, we can for instance state that the probability of a person having gene X depends on that person's mother having gene X, *and the dependence is exactly the same for all persons*.

Note that plate models are mostly useful when the variables denote properties of different objects, and there can be one-to-many or many-to-many relationships between these objects. When all variables semantically denote a property of the same object (for instance, length and weight of a person), or denote properties of different objects among which there is a one to one relationship, then plate notation is typically not needed. When there are one-to-many relationships, objects on the “many” side are typically interchangeable, which means that their relationship to the one object must be the same.

The fact that plates imply that the same factor is shared by multiple substructures is not a restriction; when multiple substructures may actually have different factors, it suffices to introduce an additional variable within the plate that indicates a parameter of the factor specification; since that variable may have different values in the different occurrences, the actual factors can be different.

Plates have been introduced *ad hoc* into graphical models. They are mostly defined by illustrations and incomplete definitions examples; a single formal and precise definition for them does not exist, though formal definitions for certain variants have been proposed [28]. While plates are very useful, their expressiveness is limited. For instance, if we would have a sequence of variables  $X_i$ ,  $i = 1, \dots, n$  where each  $X_i$  depends on  $X_{i-1}$  in exactly the same way, i.e.,  $Pr(X_1, \dots, X_n) = Pr(X_1)Pr(X_2|X_1) \cdots Pr(X_n|X_{n-1}) = Pr(X_1) \prod_{i=2}^n Pr(X_i|X_{i-1})$ ,



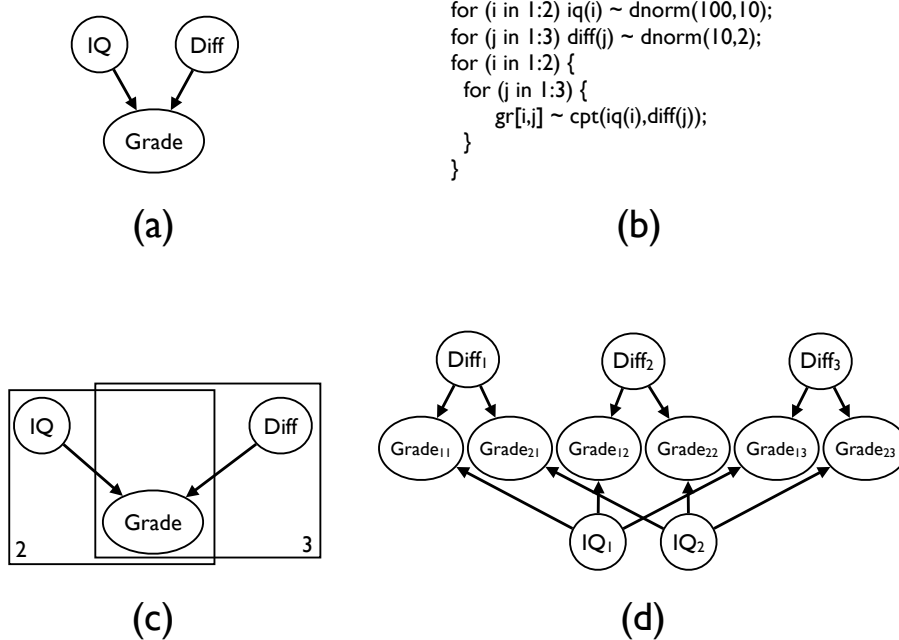
**Fig. 3.** Left: a simple Bayesian network using plate notation; right: the corresponding Bayesian network without plate notation. The right graph imposes a strictly weaker constraint: it does not indicate that the different  $Y_i$  depend on  $X$  in exactly the same way.

this cannot be expressed using plate notation. The reason is that a single variable can take two different roles in the same plate (as “parent” in one instantiation of the plate, and as “child” in another instantiation of the same plate). Further, plate models are easy to understand when plates are properly nested, but more difficult when plates can overlap (which is allowed under certain conditions). Heckerman et al. [28] introduce a variant of plates that has additional annotations that lift many of the restrictions; the simplicity of the basic plates models is then lost, however, and the formalism becomes equivalent to entity-relationship models, see later.

Together with plates, Spiegelhalter and his colleagues introduced BUGS [25], a language for defining graphical models with plates. Besides defining the structure of the model itself, the user can also indicate the format of certain factors in some detail; for instance, the conditional probability distribution of one variable given another one can be constrained to a particular family of distributions. Parameter learning is then reduced to learning the parameter of that distribution, rather than learning the conditional probability distribution in tabular format. Thus, the BUGS language is strictly more expressive than plate notation.

*Example 5.* Imagine that we have two students and three courses; each student has an IQ, each course has a difficulty level (Diff), and the grade (Gr) a student obtains for a course depends on the student’s IQ and the course’s difficulty. We could simply build a graphical model stating that grade Gr depends on IQ and Diff (Figure 4a), the parameters of which can then be learned by looking at six examples of students getting a grade for some course, but that does not take into account the fact that we know that some of these grades are really about the same student, or about the same course. Figure 4b shows (part of) an example BUGS program that does express this; Figure 4c shows the plate model that corresponds to the BUGS model (but which does not show certain information about the distributions), and Figure 4d shows the ground graphical model that

corresponds to this plate model (this ground graphical model does not show that certain conditional distributions must be equal).



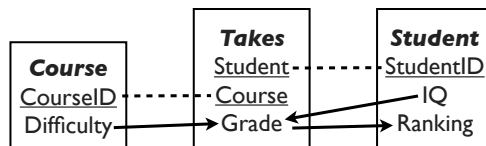
**Fig. 4.** (a) A graphical model that simply states that a Grade for an exam depends on the IQ (of the student taking that exam) and the Difficulty (of the course being examined). It does not express a certain relational structure, namely that some of these grades are obtained by the same student, or obtained for the same course. (b) a BUGS program that expresses that IQ of students and Difficulty of courses are normally distributed, that the grade obtained at an examination depends on IQ and Difficulty, and that we have 2 students who have taken 3 courses; (c) the corresponding plate model; (d) a corresponding graphical model. From (b) to (d), each consecutive model carries less information than the preceding one.

## 5.2 Approaches in the relational database setting

**Probabilistic Relational Models** Among the best known representation formalisms for statistical relational learning are probabilistic relational models or PRMs [23]. PRMs extend Bayesian networks to the relational representation used in relational databases. They model the joint probability distribution over the non-key attributes in a relational database schema. Each such attribute corresponds to a node and direct dependencies are modeled by directed edges. Such

edges can connect attributes from different entity types that are (indirectly) related (such a relationship is called a “slot chain”). Inference in PRMs occurs by constructing a Bayesian network by instantiating the PRM with the data in the database and performing the inference in the latter. To handle 1:N relationships in the Bayesian network, PRMs make use of predefined aggregation functions.

*Example 6.* Consider again the example with grades obtained depending on the student’s IQ and the course’s difficulty. Figure 5 shows a graphical representation of a PRM that corresponds to the plate model shown before.



**Fig. 5.** A probabilistic relational model structure indicating the existence of three classes of objects (Courses, Students, and instances of the Takes relation), what the attributes they have, and how these attributes depend on each other. Dashed lines indicate foreign key relationships. Arrows indicate which attributes depend on which other attributes. As the relationship between students and grades is one to many, the dependency of a student’s ranking on her grades is actually a dependency of one variable on multiple variables.

**Relational Bayesian networks** Relational Bayesian Networks [30] are another formalism for statistical relational learning; they were developed independently from PRMs and are similar to them in the sense that they also use the relational model and that the models are essentially bayesian networks. We do not discuss them in more detail here but refer to the relevant literature [30].

**Entity-relationship probabilistic models** Heckerman et al. [28] compare the expressiveness of plate models and PRMs and propose a new model, called Entity-Relationship Probabilistic Models, that generalizes both. As the name suggests, these models use the Entity-Relationship model known from relational database theory to specify the structure of probabilistic models. With respect to expressiveness, ERPMs come close to the logic-based formalisms we discuss next, while retaining a graphical, schema-like, flavor. Again, we refer to the literature [28] for a more thorough discussion.

### 5.3 Approaches in the logical setting

The integration of first order logic reasoning with probabilistic inference is a challenging goal, on which research has been conducted for several decades, with clear progress but limited convergence in terms of the practical formalisms that are in use. A particular strength of this type of approaches is that they can rely on a strong theoretical foundation; the combination of logic and probability has been studied more formally than, for instance, the expressiveness of plate models or the annotation of entity-relationship models with probabilistic information.

We first discuss a few general insights about logic and probability; next, we will discuss a number of formalisms.

#### 5.3.1 Probabilistic Logics

While both logical and probabilistic inference are well-understood, their integration is not as straightforward as it may seem. The semantics of probabilistic information in the context of first order logic can be defined in different ways. In this section we consider two dimensions along which the semantics of probabilistic logics may differ.

**Type 1 versus Type 2 semantics** Seminal work on the integration of logic and probability has been conducted by Halpern [27]. To begin with, Halpern points out that there are different types of probabilities that one can make statements about. A statement such as “the probability that a randomly chosen bird flies” is inherently ambiguous, unless we specify what kind of probability we are referring to. In what Halpern calls a Type 1 probabilistic logic, a logical variable has a distribution over its domain associated with it, and we can talk about the probability of that variable  $x$  taking a particular value. For instance, if logical variable  $x$  has a uniform distribution over the domain  $\{Tweety, Oliver, Fred, Larry, Peter\}$ , it may hold that  $Pr_x(Flies(x)) = 0.4$ ; this formula states that if we choose a random bird  $x$  from this particular distribution, there is a probability of 0.4 that  $Flies(x)$  holds.

In Type 2 logics, probabilities about possible worlds are given.<sup>4</sup> Such probabilities are most easily interpreted as a degree of belief. Thus, we might state, for instance, that there is a probability of 0.2 that Tweety flies:  $Pr(Flies(Tweety)) = 0.2$ . Note that this cannot be expressed (using the same vocabulary of predicates and constants) using a Type 1 logic: in one particular world, Tweety either flies or it does not, so  $Pr_x(Flies(Tweety))$  is either 0 or 1.

Which type of logic is most natural in a particular situation depends on the application. If we wish to describe what the probability is that  $x$  takes a particular value, given a certain (partially randomized) process for computing it, a Type 1 logic expresses this more directly. However, if we want to express

---

<sup>4</sup> A “possible world” is an assignment of truth values to all ground facts; for instance, given two propositions  $p$  and  $q$ , there are four possible worlds: one where  $p$  and  $q$  are true, one where both are false, and two where exactly one of them is true.

a certain degree of belief that a particular fact is true, a Type 2 logic is more natural. (A Type 1 logic could be used here as well, but this would require introducing a logical variable  $x$  that represents a particular world, and talking about  $Flies(Tweety, x)$  to indicate whether Tweety flies in a particular world  $x$ ; this is not a very natural way of expressing things.) The two types of logics can in principle be mixed: for instance, when we talk about the probability that a coin is fair, we talk about the (type 2) probability that the (type 1) probability of obtaining heads is 0.5. Halpern calls such a combined structure a Type 3 logic.

As an example of the kind of reasoning that is possible with a Type 3 logic, consider the following program and query:

```
0.8: Flies(Tweety).
0.2: Flies(Oliver).
0.5: Flies(Fred).
1.0: Flies(Larry).
0.5: Flies(Peter).
```

```
Bird(x) -> x ~ Unif({Tweety, Oliver, Fred, Larry, Peter}).
```

The facts are annotated with type 2 probabilities. The rule on line 6 is our way of specifying a distribution over the domain of  $x$  when  $x$  is of type Bird; it determines type 1 probabilities. Consider the query  $? - Pr(Pr_{x|Bird(x)}(Flies(x)) \geq 0.2)$ . The query asks: if we select a random world from all possible worlds, what is the (type 2) probability that for this world it holds that the (type 1) probability that a randomly chosen bird flies is at least 0.2? In this case, the answer is 1: since Larry flies in each possible world, and when choosing  $x$  there is a 0.2 probability that we choose Larry, the probability that a randomly chosen bird flies will always be at least 0.2. A general way to compute  $? - Pr(Pr_{x|Bird(x)}(Flies(x)) \leq p)$ , for  $p > 0.2$ , is to compute all possible worlds and their probability, check for each of these worlds whether the mentioned type 1 probability is at least  $p$ , and add up the probabilities of all these worlds.

Apart from introducing these logics, Halpern also shows that a complete axiomatization for them is not possible. As a result, for practical uses it is necessary to consider simpler versions of these logics. In the same way that practical logical inference systems (such as Prolog) use subsets of first order logic, practical probabilistic-logical methods will use more restrictive formalisms than Halpern's.

**Proof versus model semantics** In probabilistic logic learning, two types of semantics are distinguished [15]: the model theoretic semantics and the proof theoretic semantics. Approaches that are based on the model theoretic semantics define a probability distribution over interpretations and extend probabilistic attribute-value techniques, such as Bayesian networks and Markov networks, while proof theoretic semantics approaches define a probability distribution over proofs and upgrade, e.g., stochastic context free grammars.

*Example 7.* Consider the case where each example is a sentence in natural language. In this example, a model theoretic approach would define a probability

distribution directly over sentences. A proof theoretic approach would define a probability distribution over “proofs”, in this case possible parse trees of the sentence (each sentence may have several possible parse trees). Note that the proof theoretic view is more general, in the sense that the distribution over sentences can be computed from the distribution over proofs.

### 5.3.2 Examples of formalisms

Next, we will look at a number of different formalisms in which probabilistic logical models can be written down. They will mostly be illustrated with examples. As these formalisms sometimes express quite different types of knowledge, the corresponding examples also differ; it is difficult to define a single running example to compare all the formalisms because an example for one formalism is not necessarily suitable for illustrating the other.

Another point is that, in practice, in all these formalisms, probabilities can, but need not, be stated by the programmer; where not stated, they can be learned from data when necessary. A more difficult challenge is the learning of the program structure from data; this typically requires a search through the space of all possible model structures. Methods for learning parameters and structure have been proposed for most formalisms we mention here, but we will not go into detail about them.

All the examples below are examples of what is called *knowledge based model construction* (KBMC). This term was first introduced by Haddawy [26] and refers to the fact that a probabilistic model is constructed by means of a “program” that defines how the model should be constructed from certain knowledge available in the domain. That knowledge can be stated declaratively, or it can be hard-coded in an imperative program; we will see examples of both. In most cases, programming the model requires an understanding of the type of model that is being programmed; for instance, in formalisms based on Bayesian networks, the user is expected to know what a Bayesian network represents. Perhaps the most notable exception to this rule is Markov Logic, where the intuitive meaning of a program is relatively independent from the underlying Markov network.

**Stochastic logic programs** Stochastic logic programs (SLPs) [46] follow the proof theoretic view and upgrade stochastic context free grammars to first order logic. SLPs are logic programs with probabilities attached to the clauses such that the probabilities of clauses with the same head sum to 1.0. These numbers indicate the probability that upon a call to a predicate, this particular clause is used to resolve the calling literal. (This is similar to how in a stochastic grammar a rule  $S \rightarrow A$  is annotated with the probability that, given that a term  $S$  is encountered, this particular rule is used to rewrite it.) The probability of one particular inference chain is then computed as the product of the probabilities of the clauses that are used in the proof.



*Example 8.* The following SLP simulates the tossing of coins a variable number of times.

```
0.7: series([X|Y]) :- toss(X), series(Y).
0.3: series([]).
0.5: toss(heads).
0.5: toss(tails).
```

When the query `?-series(X)` is called, there is a 30% chance that it results in `X=[]` (i.e., the second of the two clauses for `series` is used to answer the query). There is a 70% chance that the first clause is used to answer the query, instead of the second one; in this case, the first element of the list is instantiated with `heads` or `tails` with 50% probability each, after which a recursive call repeats the process. Repeatedly calling the same query is equivalent to random sampling from a process that in the end will give the following distribution:

```
X=[] : 0.3
X=[heads]: 0.105
X=[tails]: 0.105
X=[heads,heads]: 0.03675
X=[heads,tails]: 0.03675
X=[tails,heads]: 0.03675
X=[tails,tails]: 0.03675
X=[heads,heads,heads]: 0.0128625
...
```

Note that, in general, an inference chain may also fail (this was not the case in the above example), in which case no instantiation is returned.

SLPs provide an elegant way of describing stochastic processes; executing them amounts to random sampling using these processes. When failing inference chains exist, the SLP can also be used to estimate the probability of a literal being true using Monte Carlo sampling.

SLPs can easily be used to express Type 1 probabilities. For instance, the SLP

```
0.2: bird(tweety).
0.2: bird(oliver).
0.2: bird(fred).
0.2: bird(larry).
0.2: bird(peter).
```

ensures that when the query `?- bird(X)` is called, `X` is instantiated to `tweety` or to other constants, each with a probability of 0.2.

Note that the numbers annotating the facts are not type 2 probabilities; the meaning of `0.2: bird(tweety)` is not that  $Pr(Bird(Tweety)) = 0.2$ , but  $Pr(x = Tweety|Bird(x)) = 0.2$ , which is a Type 1 probability. More specifically, the SLP defines the distribution associated with the variable `X` when `?-bird(X)` is called. Type 2 probabilities are difficult to write down in an SLP.

**Prism** The PRISM system [54] follows an approach that is somewhat similar to the SLP approach. Here, no probabilities are associated with clauses, but there is a so-called choice predicate that indicates that one of a number of alternative instantiations is chosen. In the case of PRISM the choice predicate is called `msw`, for multi-valued switch, and it instantiates a variable with a particular constant according to a given distribution, which is defined by a `set_sw` predicate. Thus, the following PRISM program and query

```
values(bird, [tweety,oliver,fred,larry,peter]).
set_sw(bird, [0.2, 0.2, 0.2, 0.2, 0.2]).
```

holds exactly the same information as the SLP shown above, and the query `?- bird(X).` instantiates the variable `X` with `tweety` in 20% of the cases.

PRISM is among the most elaborated probabilistic-logical learning systems, with a clearly defined distribution-based semantics and efficient built-in procedures for inference and learning. Further details on it can be found at the PRISM website, [sato-www.cs.titech.ac.jp/prism/](http://sato-www.cs.titech.ac.jp/prism/), which also contains pointers to the extensive literature.

**Bayesian logic programs and Logical Bayesian networks** Bayesian Logic Programs (BLPs) [34] aim at combining the inference power of Bayesian networks with that of first-order logic reasoning. Similar to PRMs, the semantics of a BLP is defined by translating it to a Bayesian network. Using this network, the probability of a given interpretation or the probability that a given query yields a particular answer can be computed.

Logical Bayesian networks [19] are a variant of BLPs in which a cleaner distinction is made between the definition of which stochastic variables exist (that is, which objects exist in the domain of discourse, and which objects have what properties), and the probabilistic inference. In other words, probabilistic inference always happens in a deterministically defined network, whereas in BLPs the network's definition can itself be a result of probabilistic inference.

The following (taken from Fierens et al. [19]) is an example of an LBN:

```
/* definition of the random variables */
random(iq(S)) <- student(S).
random(ranking(S)) <- student(S).
random(diff(C)) <- course(C).
random(grade(S,C)) <- takes(S,C).

/* definition of dependencies */
ranking(S) | grade(S,C) <- takes(S,C).
grade(S,C) | iq(S), diff(C).

/* definition of the universe */
student(john). student(pete).
course(ai). course(db).
takes(john,ai). takes(john,db). takes(pete,ai).
```

The LBN states in its first part that for each student  $s$ , a stochastic variable  $iq(s)$  and another stochastic variable  $ranking(s)$  is defined; for each course  $c$ , a variable  $diff(c)$  is defined; and for each student-course combination a variable  $grade(s, c)$  is defined. The second part states under what conditions there are direct dependencies among stochastic variables.

The first two parts of this example program together define a function  $F$  that maps interpretations to Bayesian networks. The third part states some knowledge about the world; here it is a set of ground facts, but it could be any logic program. The minimal model of this program is the interpretation that serves as an input to the function  $F$ .

Compared to PRMs, the first two parts (defining the variables and dependencies) play the role of the relational database schema. The LBN specification is less rigid in the sense that any logic program can be used to define the “schema”; for instance, if we wanted to express in the LBN that a student is only graded for a course if she is registered for examination this term, we can simply change one rule into

```
random(grade(S,C)) <- takes(S,C), registered(S).
```

Standard schema definition languages for relational databases, such as SQL, do not support such complicated schema definitions; also for PRMs it is not clear how this can be done. In this sense, LBNs are more expressive than PRMs.

In a BLP, the above program would be written as follows:

```
iq(S) | student(S).
ranking(S) | student(S).
diff(C) | course(C).
grade(S,C) | takes(S,C).
grade(S,C) | iq(S), diff(C), takes(S,C).
ranking(S) | grade(S,C), takes(S,C).
```

Note that the BLP has a simpler structure, but does not make certain details explicit. There is an essential difference between structure-determining predicates, such as *takes*, and stochastic variables, such as *grade*. The network resulting from this should contain a stochastic variable  $grade(john, ai)$ , but should not contain a stochastic variable  $takes(john, ai)$ ; rather, the *takes* predicate defines the conditions under which there should be an edge between other variables. The role of `takes(S,C)` in the BLP is very different from that of `grade(S,C)` or `iq(S)`, but this is not visible in the clauses. To compensate for this, BLPs come with a graphical interface, Balios [35], in which these additional constraints can be specified in the graphical representation of the network. BLPs as defined in Balios are more similar to LBNs. For a more extensive comparison of LBNs with PRMs and BLPs, we refer to Fierens et al. [19].

**Markov logic networks** Markov networks, also known as Markov random fields, are undirected probabilistic graphical models. In these models, there is

an edge between two nodes if and only if knowing the value of one node carries information about the other, regardless of what other evidence is given.

Markov Logic Networks (MLNs) [52] can be seen as an upgrade to first order logic of Markov networks. MLNs are defined as sets of weighted first order logic formulas. These formulas do not have to be universally true in order to be valid. They are viewed as “soft” constraints on logical interpretations: an interpretation that violates a formula is not considered impossible, it is simply considered less likely. More specifically, each ground instantiation of a formula that evaluates to false in a particular interpretation reduces the probability of that interpretation with a constant factor.

*Example 9.* A frequently used toy example in the context of Markov Logic is the following:

```
Friend(x,y) <=> Friend(y,x).  
Friend(x,y), Smokes(x) => Smokes(y).
```

The formulas indicate that the Friend relationship is symmetric, and when one person smokes, friends of this person smoke as well. In standard logic, these clauses would not be useful, because strictly speaking they are incorrect: Friend is not perfectly symmetric, and it is not the case that whenever someone smokes, all their friends necessarily smoke. In Markov logic, the logical formulas are not interpreted as universally true, but as statements that “tend to be true”, in the sense that they have few exceptions. Each formula will be assigned a weight, which can be learned from a data set; the larger a weight, the more each exception to the formula reduces the overall probability of the model, other things being equal.

Note that, while the underlying inference engine is based on Markov networks, there is nothing in the structure of the Markov logic network that shows this. The user can simply state some logical formulas that he or she believes are usually true; the Markov logic inference engine will determine weights for the formulas reflecting how strongly they hold on a given dataset, and next, be able to tell the user with what probability a certain statement is true.

The Alchemy system<sup>5</sup> implements structure and parameter learning for MLNs. It is among the most popular SRL systems at the time of writing this text.

**CP-Logic** CP-logic [67], originally called “Logic programs with annotated disjunctions” [68], differs from the other approaches in that it defines a *causal* probabilistic model. The causality is not just an interpretation that can be given to the model (and which might be correct or incorrect); the causal interpretation is by definition correct, because it is in the semantics of the model. This is very different from, for instance Bayesian networks, and it sets CP-logic apart from the other formalisms discussed here.

---

<sup>5</sup> <http://alchemy.cs.washington.edu/>

A CP-logic rule is of the form

$$h_1 : \alpha_1 \vee h_2 : \alpha_2 \vee \dots \vee h_k : \alpha_k \leftarrow b_1, b_2, \dots, b_n$$

where the  $h_i$  and  $b_j$  are literals and the  $\alpha_i$  are reals such that  $\forall i : \alpha_i \geq 0$  and  $\sum_i \alpha_i \leq 1$ . The rule specifies a part of a causal process, and states that whenever at some point the body becomes true for a particular instantiation of the logical variables, an *event* happens that *causes* at most one of the head literals to become true. (If the selected literal was already true, the event has no effect.) More specifically, one literal is drawn from the set of head literals according to the distribution specified by the  $\alpha_i$  (if  $\sum_i \alpha_i < 1$ , there is a probability of  $1 - \sum_i \alpha_i$  that nothing is selected), and the value of that literal is set to true, regardless of what it was before. Whenever an event happens that causes one literal to be selected, the outcome of this event is independent of any other such events that occurred earlier. Thus, selections in different rules, as well as selections in different instantiations of the same rule, are made independently.

*Example 10.* The following CP-logic program [43] describes how two people may go shopping and buy particular kinds of food:

```
0.3: buys(john, spaghetti) v 0.7: buys(john, chicken) <- shops(john).
0.4: buys(mary, spaghetti) v 0.6: buys(mary, fish) <- shops(mary).
0.8: shops(john).
0.5: shops(mary).
```

It states that John may decide to go buy some food today, and with a certain probability will buy spaghetti or chicken (and only one of these); similarly, Mary may buy spaghetti or fish. The rules are causal: if anything prevents John from shopping, there will be no chicken tonight.

While the structure of CP-logic programs may seem similar to that of Bayesian networks, there are several important differences. First: a Bayesian network defines a factorization of a joint distribution, not a causal structure; while arcs can be interpreted as causal, this interpretation is not part of the network's semantics. In a CP-logic program, it is. Second, the numbers we find in the conditional probability distributions of a Bayesian network are conditional probabilities; a node  $Y$  with parent  $X$  is annotated with a table that contains  $Pr(Y|X)$ . The numbers we find in a CP-logic program are not conditional probabilities; they can be equal to them, or they can be smaller. For instance, when we have a rule `0.5: y <- x`, the conditional probability of  $y$  given  $x$  is at least 0.5 (because when  $x$  is true, this alone already causes  $y$  to be true in 50% of the cases), but it can be greater because there may be other events that make  $y$  true. Third, a CP-logic program can indicate cyclic causality, while a Bayesian network cannot contain cycles. For instance, when we have two cogwheels A and B that are connected to each other, if an external cause makes cogwheel A turn, then this causes B to turn as well, but also vice versa: if an external cause makes B turn, that causes A to turn. A Bayesian network cannot express such bidirectional causality, while a CP-logic program can simply contain both `a <- b` and `b <- a` [67].

**BLOG** BLOG [44], which stands for Bayesian Logic, is yet another approach to combining probabilistic with logic inference. Special about this one is that it explicitly aims at reasoning about worlds with unknown objects, or worlds in which it is not known whether two constants actually refer to the same object or not (identity uncertainty).

*Example 11.* The following example BLOG program is taken from [44]. The program defines a stochastic process where four balls are drawn (with replacement) from an urn; we do not know how many balls are in the urn, but we do know that the balls that have been put in the urn were selected randomly from a population with 50% blue and 50% green balls.

```

type Color; type Ball; type Draw;

random Color TrueColor(Ball);
random Ball BallDrawn(Draw);
random Color ObsColor(Draw);

guaranteed Color Blue, Green;
guaranteed Draw Draw1, Draw2, Draw3, Draw4;

#Ball ~ Poisson[6]();
TrueColor(b) ~ TabularCPD[[0.5,0.5]]();
BallDrawn(d) ~ Uniform({Ball b});

ObsColor(d)
  if (BallDrawn(d) != null) then
    ~ TabularCPD[[0.8, 0.2], [0.2, 0.8]] (TrueColor(BallDrawn(d)));

```

The first line defines three types of objects; the extension of each type is a set of objects, the size of which is not specified at this point. The `random` statements define random variables associated with each object, and define the values that these random variables can take; for instance, with each ball a variable `TrueColor` is associated, and this variable takes on a value of type `Color`. Next, the extensions of `Color` and `Draw` are specified: we guarantee that there are exactly two colors (Blue and Green), and four draws. The extension of `Ball` is not known so precisely: its cardinality, i.e., the number of balls, is unknown but a probability distribution is given for it (Poisson distribution with parameter 6). Given a ball, its `TrueColor` is Blue or Green with a probability of 0.5 each; given a draw, the ball that is drawn is one from the extension of `Ball`, drawn uniformly. Finally, the observed color of a ball is the same as its true color in 80% of the cases; in 20% of the cases the color is observed incorrectly.

As can be seen in the example, BLOG strives explicitly at defining *sets* of objects (the elements of which may remain anonymous), rather than individual objects, as is the case in for instance LBNs (where the universe is explicitly defined and each object gets its own name). While most methods work with a

fixed universe, and define a distribution over interpretations for this universe, a BLOG model extends this principle by defining a distribution over universes.

**ProbLog** By now the reader will be convinced that many different approaches to statistical relational learning exist, and he or she may wonder why this is the case. Part of the answer probably lies in the fact that (on the one hand) a need is felt for the kind of expressiveness that these formalisms offer, but (on the other hand) this need is application-driven, and thus most researchers have developed a formalism that is suitable for the kind of applications *they* were thinking about. This has led to a variety of formalisms that share many properties, but also have their own specificity, which is often desired for the context they are used in. This is also visible in the examples chosen to illustrate the formalisms: these differ quite strongly.

Seeing the variety as well as the commonalities in all these formalisms, De Raedt et al. [13] argue that there is a need for an underlying programming language, in which the other formalisms could be implemented, but which would itself offer important functionality for efficient probabilistic inference. To this aim, they propose the probabilistic-logical programming language ProbLog. ProbLog is a conceptually very simple extension of the well-known logic programming language Prolog. A ProbLog program consists of a set of definite Horn Clauses, just like a standard Prolog program, but each fact is additionally (and optionally) annotated with a number that expresses the probability that this fact is true.

The following is an example of a ProbLog program:

```
path(X,Y) :- edge(X,Z), path(Z,Y).
edge(1,2).
0.4: edge(2,3).
0.2: edge(3,4).
0.3: edge(1,3).
```

The query `?- path(1,4)` results in a probability distribution over **yes** and **no**, with **yes** having a probability of (we shorten **path** and **edge** to *p* and *e*):

$$Pr(p(1,4)) = Pr(p(1,3))Pr(e(3,4))$$

with

$$\begin{aligned} Pr(p(1,3)) &= Pr(e(1,2))Pr(e(2,3)) + Pr(e(1,3)) - Pr(e(1,2))Pr(e(2,3))Pr(e(1,3)) \\ &= Pr(e(2,3)) + Pr(e(1,3)) - Pr(e(2,3))Pr(e(1,3)) \quad (\text{as } Pr(p(1,2)) = 1) \\ &= Pr(e(2,3)) + (1 - Pr(e(2,3)))Pr(e(1,3)) \\ &= 0.4 + 0.6 * 0.3 = 0.58. \end{aligned}$$

When the graph becomes more complex than this example graph, the complexity of the calculations rises quickly. This is in part due to the inclusion-exclusion principle, which states that

$$Pr\left(\bigcup_i A_i\right) = \sum_i Pr(A_i) - \sum_{i \neq j} Pr(A_i \cap A_j) + \sum_{i \neq j, k} Pr(A_i \cap A_j \cap A_k) - \dots \pm Pr\left(\bigcap_i A_i\right)$$

which in the case of independent  $A_i$  becomes

$$Pr(\bigcup_i A_i) = \sum_i Pr(A_i) - \sum_{\substack{j \\ \neq i}} Pr(A_i)Pr(A_j) + \sum_{\substack{j,k \\ \neq i}} Pr(A_i)Pr(A_j)Pr(A_k) - \dots \pm \prod_i Pr(A_i)$$

The size of this formula increases exponentially with the number of  $A_i$ , and computing path probabilities in graphs is a generalization of the above calculation. Thus, computing this probability easily becomes intractable for large graphs. However, much effort has been spent on compiling such structures into more efficient representations, many of which are variants of the so-called binary decision diagrams (BDDs). ProbLog uses a similar translation to render the computation of probabilities more tractable. Besides this, the ProbLog engine implements many other ideas that improve efficiency. All together, they make it possible to answer queries as efficiently as possible. The idea behind ProbLog is that it could be used to implement engines for other formalisms, which will then automatically inherit all these efficient implementations.

For further details about ProbLog we refer to the literature [36,4].

#### 5.4 Other approaches

More specific statistical learning techniques, such as Naïve Bayes and Hidden Markov Models, have also been upgraded to the relational setting [22,59]. While the above formalisms are mostly logic or relational database oriented, other types of languages have been proposed; IBAL [49], for instance, is a functional probabilistic language, while CLP(BN) [53] makes use of constraint logic programming to integrate probabilistic with logical inference. A much more complete and in-depth overview of approaches is presented by Getoor and Taskar [24].

## 6 General remarks and Challenges

Many challenges remain in the area of statistical relational learning. We list a few that are currently attracting a significant amount of interest from researchers.

### 6.1 Understanding commonalities and differences

A large number of formalisms for statistical relational learning exist. This situation is sometimes referred to as “the alphabet soup”: we have BLOG, BLPs, BUGS, CLP(BN), CP-logic, IBAL, ICL, LBNs, MLNs, PRISM, PRMs, ProbLog, RBNs, SLPs, . . . It seems a natural goal to try to merge all these formalisms into a single one (or at least a few ones) that subsumes all of them. Yet, this goal seems difficult to achieve, and the validity of the goal has been challenged: it is possible that depending on the task, one formalism is much more suitable than another, so why try to merge them into one? Still, even if such a grand unification is not necessarily desirable, it seems useful to understand better the differences between all these approaches. Currently, such an understanding exists to some extent, but it remains largely anecdotal and incomplete. Several



researchers have shown how to translate programs from one formalism to another, showing equivalence or subsumption between some formalisms. Yet, such translations have to be interpreted with caution, as there are several levels of “equivalence”: one can define equivalence in terms of the actual models that can be learned, in terms of the model structures, or in terms of the sets of model structures that can be given by the user as a bias for the learner; models can be interpreted as (constraints over) joint distributions, or as functions that map a logical interpretation onto such a (constraint over a) distribution. A single well-understood framework for comparing formalisms does not appear to exist at this moment. Finally, there is the issue of user-friendliness: how easily can a user write down certain background knowledge, and how easily can that user interpret the models? Again, “interpretation” is a broad term here. For instance, in Markov Logic, clauses are given weights that tell us how likely the clauses are to be violated, but there is no simple connection between these weights and probabilities, which contrasts with the situation in, for instance, BLPs. On the other hand, probabilities of specific facts can of course be inferred by the model whenever necessary.

Earlier in this chapter we have discussed differences in relational learners in terms of the features that they implicitly construct. A comparison between SRL systems from this point of view would be one way in which additional insight can be gained. Such a comparison would indicate to what extent the different SRL formalisms are truly relational; as said before, a relational learner that constructs only a small set of features could be said to be “less relational”, and is in that sense less expressive.

There has been a number of practical comparisons of SRL systems. A large number of approaches is compared, from a user’s point of view (how easy is it to model a particular problem using a particular SRL approach), by Bruynooghe et al. [3] and Taghipour et al. [58]. In both papers the authors conclude that simple problems can still be hard to model with even the most advanced SRL approaches available today.

## 6.2 Parameter learning and structure learning

The learning of probabilistic graphical models involves two distinguished tasks: parameter learning and structure learning. *Parameter learning* is the easiest task. Given a model structure (i.e., a directed or undirected graph, in the case of Bayesian or Markov networks; a set of first order logic clauses in the case of Markov logic; etc.), the task is to fit the model to the data, i.e., determine the parameter settings for which an optimal fit is obtained. *Structure learning* is more difficult. It involves determining an optimal model structure (i.e., determining the optimal graph structure, determining the optimal set of clauses, etc.). This in itself often involves a search through the model space, where each model is individually evaluated by fitting it to the data (i.e., via parameter learning) and measuring how good the fit is. However, the structure learning step may exploit additional background knowledge that the user has about the likely structure of the model.

Parameter learning is a relatively standard task by now. Structure learning, on the other hand, needs to be implemented differently depending on the formalism that is being used; for instance, since the syntax of CP-logic programs is quite different from that of Markov logic networks, quite different structure learning approaches are required. For Markov networks, Richardson and Domingos [52] show how the structure can be determined by making use of the ILP system Claudien [12] as an auxiliary system. Meert and Blockeel [43] show how the structure of acyclic CP-logic programs can be learned by turning them into equivalent Bayesian networks that contain one latent variable per CP-logic rule and the structure of which is constrained in a particular way; they then show how standard techniques for learning the structure of Bayesian networks can be adapted to ensure the resulting networks obey these structural constraints. Structure learning methods have been proposed for many other formalisms as well [24,20].

### 6.3 Scalability

Probabilistic inference in graphical models is, in the general case, NP-hard: roughly speaking, its computational complexity increases exponentially in the size of the network. While this has always been an issue in probabilistic inference, it is even more so in statistical relational learning. The size of the network generally depends on the size of the domain, which may be very large. For instance, consider the “Friends and Smokers” example from the discussion on Markov logic. There is a stochastic variable for each pair of persons  $(x, y)$ ; when we are talking about a few thousand persons, this means there will be millions of variables in the ground network, interconnected in complex ways. Clearly, inference in this ground network will be challenging. An important approach towards alleviating this problem is the concept of *lifted inference*.

The term “lifted inference” refers to performing inference on a higher level of abstraction than the ground network. A crucial property that lifted inference methods rely on is that of indistinguishability. Sometimes two stochastic variables are exactly equivalent with respect to what is known about them. This can be true in ordinary networks, but it is much more often the case in ground models that have been generated by first-order models. For instance, in the Markov logic network for Friends and Smokers, suppose Bart has one friend who smokes and one who does not, and these do not know each other; and suppose Lisa is in exactly the same situation. If nothing else is known, then whatever we can infer about Bart we can also infer about Lisa: all probabilities will be equal.

The idea of lifted inference is similar to what is used in logic programming: in the Prolog programming language, for instance, inference is done on the level of variables, rather than constants, insofar possible. Given the rule  $q(X) :- p(X)$ , a Prolog engine can infer from  $p(a)$  that  $q(a)$  holds (i.e., apply the rule to a specific case), but it can also infer from  $p(X)$  that  $q(X)$  holds (regardless of what  $X$  is). Inference is done on the level of universally quantified variables where possible, and on the ground level where needed. Constraint logic programming provides a middle ground by allowing inference that keeps track of sets of ground

instantiations for which the current inference could be made (and these sets may be defined extensionally, by listing their elements, or intensionally, using constraints). This allows for much more efficient inference, compared to reasoning only on the ground level.

The same principle can be used in probabilistic-logical models. If we write

```
0.5: p(a) .
0.5: p(b) .
0.5: p(c) .
```

```
0.5: q(X) <- p(X) .
r(X,Y) <- p(X),q(Y) .
```

then it is clear that  $q(a)$ ,  $q(b)$  and  $q(c)$  must all have the same probability of being true (0.25), and similarly,  $Pr(r(X, Y)) = 0.125$  whenever  $X \in \{a, b, c\}$  and  $Y \in \{a, b, c\}$ . Clearly, we can compute probabilities on the level of (extensionally or intensionally defined) sets of ground literals, rather than on the level of individual literals.

Probabilistic inference is much more complicated than pure logical inference, however, and lifting it to the first-order context is a challenge that is still far from solved. It does have a large potential towards more efficient inference in statistical relational learning. Seminal work in this area was performed by Poole [50], and in the ensuing decade many other authors took up the challenge.

## 7 Recommended Reading

Starting out from a general description of relational learning and how it differs from standard learning, we have discussed neural-network based approaches to relational learning, and statistical relational learning. A much more detailed treatment of all these topics is available in several reference works. Directly relevant references to the literature include the following. A comprehensive introduction to ILP can be found in De Raedt's book [10] on logical and relational learning, or in the collection edited by Džeroski and Lavrač [18] on relational data mining. Learning from graphs is covered by Cook and Holder [7]. Džeroski and Lavrač [18] is also a good starting point for reading about multi-relational data mining, together with research papers on multi-relational data mining systems. Statistical relational learning in general is covered in the collection edited by Getoor and Taskar [24], while De Raedt and Kersting [15] and De Raedt et al. [14] present overviews of approaches originating in logic-based learning.

## Acknowledgements

This chapter builds on earlier publications written in close collaboration with Werner Uwents, Jan Struyf, and Maurice Bruynooghe. The author thanks Daan Fierens and an anonymous reviewer for valuable comments.

## References

1. H. Blockeel and M. Bruynooghe. Aggregation versus selection bias, and relational neural networks. In *IJCAI-2003 Workshop on Learning Statistical Models from Relational Data, SRL-2003, Acapulco, Mexico, August 11, 2003*, 2003.
2. I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1986.
3. M. Bruynooghe, B. De Cat, J. Drijskonigen, D. Fierens, J. Goos, B. Gutmann, A. Kimmig, W. Labeeuw, S. Langenaken, N. Landwehr, W. Meert, E. Nuyts, R. Pellegrims, R. Rymenants, S. Segers, I. Thon, J. Van Eyck, G. Van den Broeck, T. Vangansewinkel, L. Van Hove, J. Vennekens, T. Weytjens, and L. De Raedt. An exercise with statistical relational learning systems. In *Proceedings of the 6th International Workshop on Statistical Relational Learning*, 2009.
4. M. Bruynooghe, T. Mantadelis, A. Kimmig, B. Gutmann, J. Vennekens, G. Janssens, and L. De Raedt. Problog technology for inference in a probabilistic first order logic. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 719–724. IOS Press, 2010.
5. W. Buntine. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, 2:159–225, 1994.
6. D.J. Cook and L.B. Holder. Substructure discovery using minimum description length and background knowledge. *J. Artif. Intell. Res. (JAIR)*, 1:231–255, 1994.
7. D.J. Cook and L.B. Holder. *Mining Graph Data*. Wiley, 2007.
8. A.S. d’Avila Garcez and G. Zaverucha. The connectionist inductive learning and logic programming system. *Appl. Intell.*, 11(1):59–77, 1999.
9. L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.
10. L. De Raedt. *Logical and Relational Learning*. Springer, 2008.
11. L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.
12. L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26(2-3):99–146, 1997.
13. L. De Raedt, B. Demoen, D. Fierens, B. Gutmann, G. Janssens, A. Kimmig, N. Landwehr, T. Mantadelis, W. Meert, R. Rocha, V. Santos Costa, I. Thon, and J. Vennekens. Towards digesting the alphabet-soup of statistical relational learning. In *Proceedings of the NIPS\*2008 Workshop Probabilistic Programming*, pages 1–3, 2008.
14. L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton. *Probabilistic Inductive Logic Programming*. Springer, 2008.
15. L. De Raedt and K. Kersting. Probabilistic logic learning. *SIGKDD Explorations*, 5(1):31–48, 2003.
16. L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
17. S. Džeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Mach. Learn.*, 43:7–52, 2001.
18. S. Džeroski and N. Lavrač, editors. *Relational Data Mining*. Springer, 2001.
19. D. Fierens, H. Blockeel, M. Bruynooghe, and J. Ramon. Logical bayesian networks and their relation to other probabilistic logical models. In Kramer and Pfahringer [38], pages 121–135.
20. D. Fierens, J. Ramon, M. Bruynooghe, and H. Blockeel. Learning directed probabilistic logical models: Ordering-search versus structure-search. In Joost N. Kok,

- Jacek Koronacki, Ramon López de Mántaras, Stan Matwin, Dunja Mladenic, and Andrzej Skowron, editors, *ECML*, volume 4701 of *Lecture Notes in Computer Science*, pages 567–574. Springer, 2007.
21. P. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacophore discovery using the inductive logic programming system Progol. *Mach. Learn.*, 30:241–270, 1998.
  22. P.A. Flach and N. Lachiche. Naive bayesian classification of structured data. *Machine Learning*, 57(3):233–269, 2004.
  23. N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In Thomas Dean, editor, *IJCAI*, pages 1300–1309. Morgan Kaufmann, 1999.
  24. L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 2007.
  25. W.R. Gilks, A. Thomas, and D.J. Spiegelhalter. A language and program for complex bayesian modelling. *The Statistician*, 43:169–178, 1994.
  26. P. Haddawy. Generating bayesian networks from probability logic knowledge bases. In Ramon López de Mántaras and David Poole, editors, *UAI*, pages 262–269. Morgan Kaufmann, 1994.
  27. J.Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46:311–350, 1990.
  28. D. Heckerman, C. Meek, and D. Koller. Probabilistic entity-relationship models, prms, and plate models. In *Introduction to Statistical Relational Learning*, pages 201–238. MIT Press, 2007.
  29. T. Horváth, J. Ramon, and S. Wrobel. Frequent subgraph mining in outerplanar graphs. In *Proc. of the 12th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pages 197–206, 2006.
  30. M. Jaeger. Relational bayesian networks. In *UAI '97: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence, August 1-3, 1997, Brown University, Providence, Rhode Island, USA*, pages 266–273. Morgan Kaufmann, 1997.
  31. D. Jensen and J. Neville. Linkage and autocorrelation cause feature selection bias in relational learning. In *Proc. of the 19th Int'l Conf. on Machine Learning*, pages 259–266, 2002.
  32. D. Jensen, J. Neville, and B. Gallagher. Why collective inference improves relational classification. In *Proc. of the 10th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pages 593–598, 2004.
  33. A. Karalić and I. Bratko. First order regression. *Machine Learning*, 26:147–176, 1997.
  34. K. Kersting. *An Inductive Logic Programming Approach to Statistical Relational Learning*. IOS Press, 2006.
  35. K. Kersting and U. Dick. Balios - the engine for bayesian logic programs. In *In Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD-2004)*, pages 549–551, 2004.
  36. A. Kimmig and L. De Raedt. Local query mining in a probabilistic prolog. In Craig Boutilier, editor, *IJCAI*, pages 1095–1100, 2009.
  37. D. Koller, N. Friedman, L. Getoor, and B. Taskar. Graphical models in a nutshell. In *Introduction to Statistical Relational Learning*, pages 13–55. MIT Press, 2007.
  38. S. Kramer and B. Pfahringer, editors. *Inductive Logic Programming, 15th International Conference, ILP 2005, Bonn, Germany, August 10-13, 2005, Proceedings*, volume 3625 of *Lecture Notes in Computer Science*. Springer, 2005.
  39. M.-A. Krogel, S. Rawles, F. Železný, P. Flach, N. Lavrač, and S. Wrobel. Comparative evaluation of approaches to propositionalization. In *Proc. of the 13th Int'l Conf. on Inductive Logic Programming*, pages 194–217, 2003.

40. M.-A. Krogel and S. Wrobel. Transformation-based learning using multirelational aggregation. In C. Rouveirol and M. Sebag, editors, *ILP*, volume 2157 of *Lecture Notes in Computer Science*, pages 142–155. Springer, 2001.
41. J.D. Lafferty, A. McCallum, and F.C.N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In Carla E. Brodley and Andrea Pohoreckyj Danyluk, editors, *ICML*, pages 282–289. Morgan Kaufmann, 2001.
42. J.W. Lloyd. *Logic for Learning*. Springer, 2003.
43. W. Meert, J. Struyf, and H. Blockeel. Learning ground CP-Logic theories by leveraging bayesian network learning techniques. *Fundam. Inform.*, 89(1):131–160, 2008.
44. B. Milch, B. Marthi, S.J. Russell, D. Sontag, D.L. Ong, and A. Kolobov. Blog: Probabilistic models with unknown objects. In Leslie Pack Kaelbling and Alessandro Saffioti, editors, *IJCAI*, pages 1352–1359. Professional Book Center, 2005.
45. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
46. S. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996.
47. S. Nijssen and J.N. Kok. The gaston tool for frequent subgraph mining. *Electr. Notes Theor. Comput. Sci.*, 127(1):77–87, 2005.
48. C. Perlich and F.J. Provost. Aggregation-based feature invention and relational concept classes. In Lise Getoor, Ted E. Senator, Pedro Domingos, and Christos Faloutsos, editors, *KDD*, pages 167–176. ACM, 2003.
49. A. Pfeffer. The design and implementation of ibal: A general-purpose probabilistic programming language. Technical Report TR-12-05, Harvard University, 2005.
50. D. Poole. First-order probabilistic inference. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 985–991. Morgan Kaufmann, 2003.
51. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
52. M. Richardson and P. Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2):107–136, 2006.
53. V. Santos Costa, D. Page, M. Qazi, and J. Cussens. Clp(bn): Constraint logic programming for probabilistic knowledge. In Christopher Meek and Uffe Kjærulff, editors, *UAI*, pages 517–524. Morgan Kaufmann, 2003.
54. T. Sato and Y. Kameya. PRISM: A symbolic-statistical modeling language. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI97)*, pages 1330–1335, 1997.
55. D.J. Spiegelhalter. Bayesian graphical modelling: a case-study in monitoring health outcomes. *Applied Statistics*, 47:115–134, 1998.
56. J. Struyf and H. Blockeel. Relational learning. In C. Sammut and G. Webb, editors, *Encyclopedia of Machine Learning*, pages 851–857. Springer, 2010.
57. P. Tadepalli, R. Givan, and K. Driessens. Relational reinforcement learning: An overview. In *Proc. of the ICML’04 Wshp. on Relational Reinforcement Learning*, pages 1–9, 2004.
58. N. Taghipour, D. Fierens, and H. Blockeel. Probabilistic logical learning for bi-clustering: A case study with surprising results. CW Reports CW597, Department of Computer Science, K.U.Leuven, October 2010.
59. I. Thon, N. Landwehr, and L. De Raedt. A simple model for sequences of relational state descriptions. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *ECML/PKDD (2)*, volume 5212 of *Lecture Notes in Computer Science*, pages 506–521. Springer, 2008.

60. G.G. Towell and J.W. Shavlik. Knowledge-based artificial neural networks. *Artif. Intell.*, 70(1-2):119–165, 1994.
61. W. Uwents. *Learning complex aggregate features with relational neural networks*. PhD thesis, Katholieke Universiteit Leuven, 2011. Forthcoming.
62. W. Uwents and H. Blockeel. Classifying relational data with neural networks. In Kramer and Pfahringer [38], pages 384–396.
63. W. Uwents and H. Blockeel. A comparison between neural network methods for learning aggregate functions. In Jean-François Boulicaut, Michael R. Berthold, and Tamás Horváth, editors, *Discovery Science*, volume 5255 of *Lecture Notes in Computer Science*, pages 88–99. Springer, 2008.
64. W. Uwents and H. Blockeel. Learning aggregate functions with neural networks using a cascade-correlation approach. In F. Zelezný and N. Lavrac, editors, *ILP*, volume 5194 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2008.
65. W. Uwents, G. Monfardini, H. Blockeel, M. Gori, and F. Scarselli. Neural networks for relational learning: An experimental comparison. *Machine Learning*, 82:315–349, 2011.
66. A. Van Assche, C. Vens, H. Blockeel, and S. Džeroski. First order random forests: Learning relational classifiers with complex aggregates. *Machine Learning*, 64(1-3):149–182, 2006.
67. J. Vennekens, M. Denecker, and M. Bruynooghe. Cp-logic: A language of causal probabilistic events and its relation to logic programming. *TPLP*, 9(3):245–308, 2009.
68. J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 431–445. Springer, 2004.
69. C. Vens, J. Ramon, and H. Blockeel. Refining aggregate conditions in relational learning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *PKDD*, volume 4213 of *Lecture Notes in Computer Science*, pages 383–394. Springer, 2006.
70. T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explorations*, 5(1):59–68, 2003.
71. X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724. IEEE Computer Society, 2002.
72. X. Yin, J. Han, J. Yang, and P.S. Yu. Efficient classification across multiple database relations: A CrossMine approach. *IEEE Trans. Knowl. Data Eng.*, 18(6):770–783, 2006.